The Comparison of Different Deep Learning Models For Facial Expression Recognition

The George Washington University

DATS 6203 Machine Learning II
Group 1
Xiaochi Li, Liwei Zhu, Jia Chen
Instructor: Professor Amir Jafari
December 2018

Outline

1. Introduction

2. Data Description

3. Deep Network and Training Algorithm

4. Experimental Setup

5. Results

6. Application: Real time face expression recognition

7. Summary and Conclusions

8. References

1. Introduction

Facial expression is a primary way to convey emotions and intentions of human beings. Detecting facial expression promotes communication between individuals. With the development of technology, facial expression recognition has attracted special research attention. A large volume of studies has been focused on "automatic facial expression analysis because of its practical importance in sociable robotics, medical treatment, driver fatigue surveillance, and many other human-computer interaction systems"[1]

In the field of machine learning, facial expression recognition has been well developed due to the improvement of database and deep learning techniques. Compared to traditional laboratory-controlled database such as the Extended CohnKanade (CK+) and the MMI database, new web database includes larger amount of expression pictures from real world, thus providing sufficient data for deep learning network. For example, the Real-world Affective Face Database (RAF-DB) including 29,672 images that annotated into 6 basic expressions[2] plus neutral and 12 compound expressions[1]. Same as web database, EmotioNet includes 1,000,000 images that are classified into 23 basic expressions or compound expressions. Another contributing factor is "increased chip processing abilities and well-designed network architecture" [1], leading facial expression recognition to deep learning method.

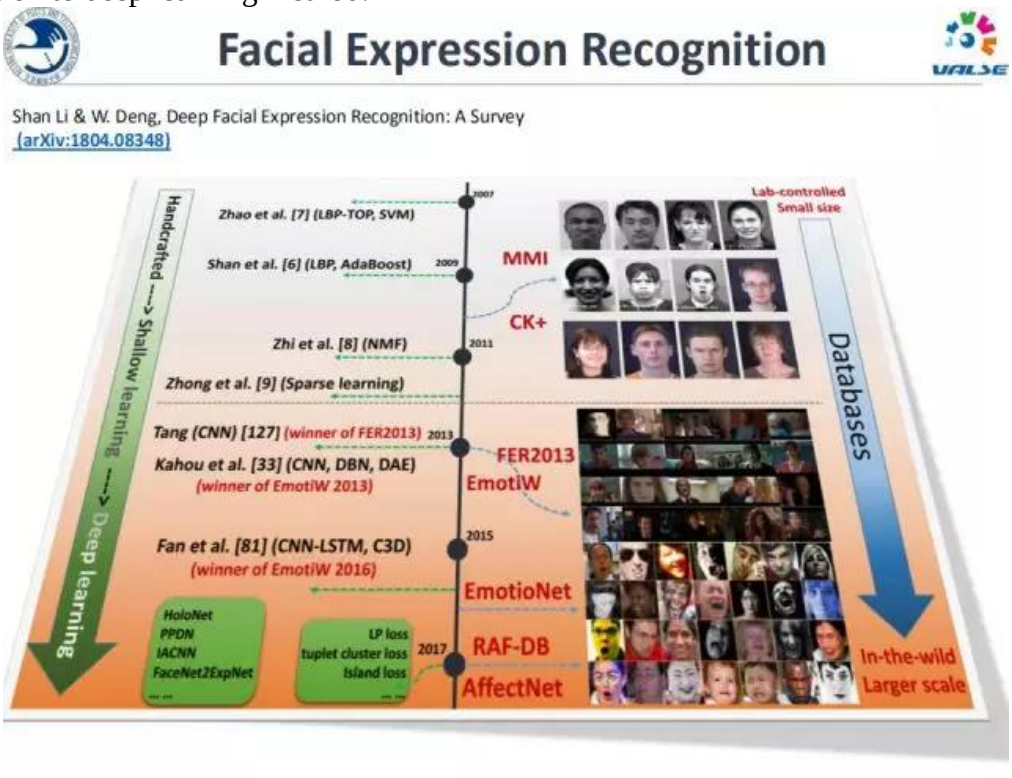


Figure 1. The development of Facial Expression Databases [1]

[1] Li, S., & Deng, W. (2018). Deep Facial Expression Recognition: A Survey. *arXiv preprint arXiv:1804.08348*.

[2] Ekman, P., & Friesen, W. V. (1971). Constants across cultures in the face and emotion. *Journal of personality and social psychology*, *17*(2), 124.

This study aims to train a deep facial expression recognition model that can classify given facial image into one of seven classes. We select the Real-world Affective Face Database (RAF-DB) as data. Different models (CNN and MLP) are implemented using TensorFlow (Keras API). In the process of optimize model to get better accuracy, we also explore how hyperparameters influence the performance of network. Finally, the performances of these models are compared. The rest of this paper is organized as follows. Section 2 describes database. Section 3 provides background information about networks and algorithms used in this project. Section 4 explains how we pre-process data and adjust architectures and parameters. Section 5 discusses study results and section 6 summarizes this project.

2. Data Description

As mentioned in last section, Real-world Affective Face Database (RAF-DB)[3] is used to train and test neural networks in this project. RAF-DB contains 29,672 real-world facial images from Internet. These images are divided into two subsets: single-label subset, including seven classes of basic emotions, and two-tabs subset, adding twelve classes of compound emotions. In this study, we only used single-label subset, which contains 15,339 images that are annotated into seven types of labels: "Surprised", "Fearful", "Disgusted", "Happy", "Sad", "Angry", and "Neural". Furthermore, single-label subset is split into two groups: 12,271 training images and 3,068 testing images. It is worth mentioning that RAF-DB adopts crowd-sourcing method to annotate images. 315 annotators label images into one of seven basic emotions, and each image is annotated around 40 times. As a result, RAF-DB is relatively reliable.

RAF-DB has been applied to train various networks such as convolutional neural network (CNN), deep belief network (DBN), and recurrent neural network (RNN). Li S. and Deng W. proposed "a new deep locality-preserving convolutional neural network (DLP-CNN) method that aims to enhance the discriminative power of deep features by preserving the locality closeness while maximizing the inter-class scatter"[4]. The accuracy of this method is 74.4%.

[3] Li, S., Deng, W., & Du, J. (2017, July). Reliable crowdsourcing and deep locality-preserving learning for expression recognition in the wild. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on* (pp. 2584-2593). IEEE.

[4] Li, S., & Deng, W. (2019). Reliable crowdsourcing and deep locality-preserving learning for unconstrained facial expression recognition. *IEEE Transactions on Image Processing*, *28*(1), 356-370.

3. Deep Network and Training Algorithm

3.1 Framework: TensorFlow (Keras API)

All the models in this project are implemented by Keras. It is an open source neural network library written in Python. With the development of deep neural network, adjusting parameters has been becoming a more and more difficult task when optimizing network. Keras makes it easier by being user friendly, modular, and extensible. "Keras contains numerous implementations of commonly used neural network building blocks such as layers, objectives, activation functions, optimizes, and a host of tools to make working with image and text data easier". Keras supports many neural networks such as MLP and CNN.

3.2 Multilayer perceptron

The first deep neural network used in this project s multilayer perceptron (MLP). MLP is a sequential artificial neural network. As shown in figure 1, in general, MLP is composed of three parts: input, hidden layer, and output layer. There can be more than one layer inside of hidden layer. There must be transfer function for each layer. After dot product and transfer function of first layer, the output of first layer becomes the input of second layers. In output layer, the output of hidden layer will be classified predicted or classified. MLP utilizes a supervised learning technique called backpropagation for training.

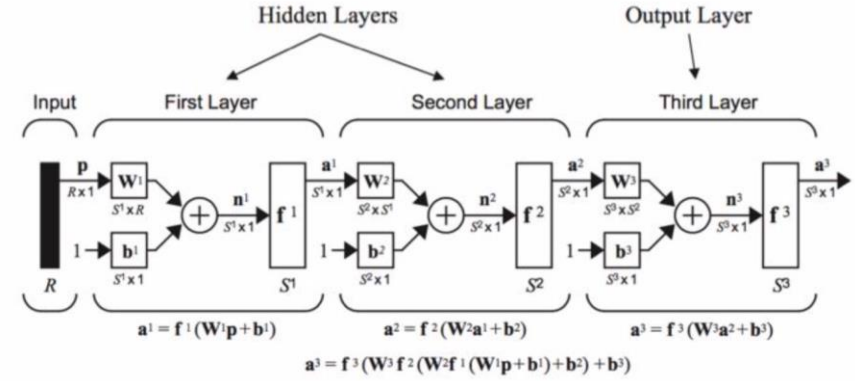


Figure 2. Typical architecture of MLP

3.3 Convolution neural network

The second deep neural network used in this project is convolution neural network (CNN). CNN is mostly applied to images recognition. CNN is made of three kinds of layers: convolutional layer, pooling layer, and fully connected layer. As shown is figure 2, the convolutional layer contain a set of kernels to convolve through the whole picture, thus each kernel generate one feature map. One characteristic of CNN is sharing weight (kernel), different from MLP. The convolutional layer is followed by pooling layer, which reduces the size of feature map and computational cost. There are two common method called average pooling and max pooling. The last layer of CNN is fully

connected layer. Due to the input of Softmax must be vector, matrix needs to be converted to vector by fully connected layer.

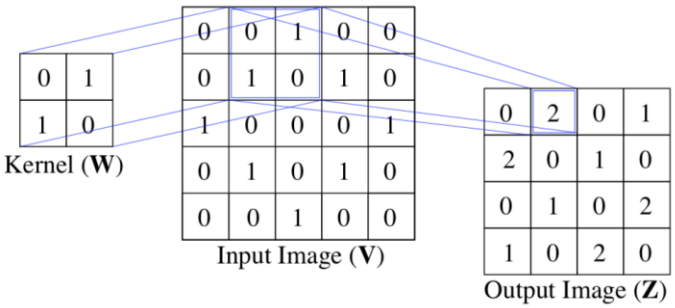


Figure 3. Kernel convolves through image

3.4 Light VGG net: One CNN structure inspired by VGGnet

After 1994 when one of the very first convolutional neural network, LeNet5 gives the most basic structures of a convolutional network, many networks follow the path of LeNet5 and evolving to becoming more powerful and complicated.

One of the most popular CNN structure and the winner of ImageNet Large Scale Visual Recognition Challenge(2014) is the VGG networks introduced by Karen Simonyan and Andrew Zisserman of University of Oxford in 2014. [5] This model followed the pattern of LeNet5 and inherited the useful features from AlexNet[6], such as ReLU transfer function and dropout technique.

The great advantage of VGG was the insight that multiple 3x3 convolutions in sequence can emulate the effect of larger receptive fields while increase the depth of the network to learn more patterns.

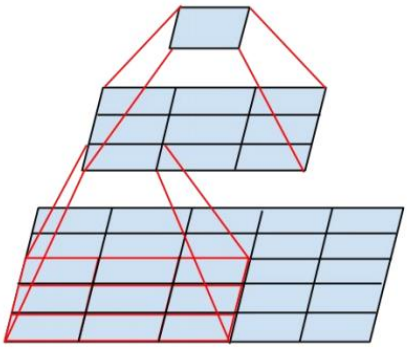


Figure 4. Use multiple 3x3 convolution

As seen in the structure plot, the VGGnet has 5 convolution blocks, which has the same structure: two 3x3 convolution layer and one max pooling layer. And followed by one

[5] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[6] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

fully connected layer and softmax layer.

The original purpose of VGGnet is to be trained on the ImageNet and classify 1000 categories of objects from 224x224x3 input. And it's more complicated than our purpose: to classify 7 expression labels from 100x100x3 input. So, we decided to simplify the structure of original VGGnet to increase its speed and avoid overfitting while keep using multiple 3x3 convolution layer in sequence and increase the number of kernels as the network goes deeper.

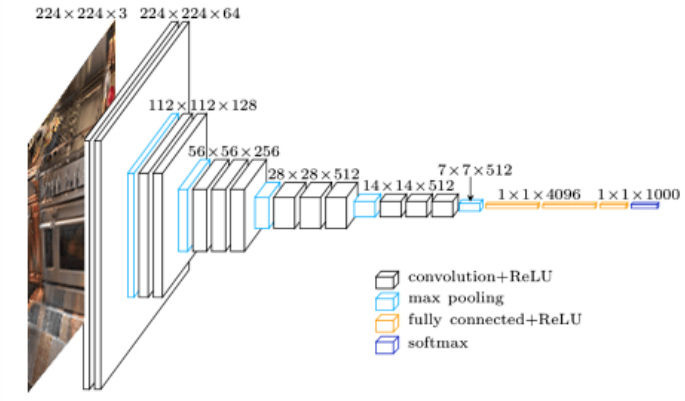


Figure 5. Structure of VGG16

4. Experimental Setup

4.1 Data pre-processing

Before building models, let us review the data quickly. We select the single-label subset of RAF-DB. This subset contains 15,339 facial images, 12,271 for training and 3068 for testing. All the images are annotated into one of seven basic emotions: "Surprise", "Fear", "Disgust", "Happiness", "Sadness", "Anger", and "Neural".

The data archive contains a zip file, "aligned.zip", and a txt file, "list_patition_label.txt". All the images are contained in "aligned.zip" and named in the format of "train_XXXXX_aligned.jpg" or "test_XXXX_aligned.jpg". Images are cropped, repositioned and resized to 100*100(RGB). In the txt file, expression label for each image is marked separately by each row as shown in figure 3. Digits from 1 to 7 represent "Surprise", "Fear", "Disgust", "Happiness", "Sadness", "Anger", and "Neural" respectively.

```
1  train_00001.jpg 5
2  train_00002.jpg 5
3  train_00003.jpg 4
4  train_00004.jpg 4
5  train_00005.jpg 5
```

Figure 6. list_patition_labels.txt

As the first step of data pre-processing (implemented in V2_helper.py), create two

dictionaries that record image name and corresponding label for training set and testing set. It is important to set the image name same as the name in aligned.zip file as seen from figure 4.

```
def label_loader(label_file):
    train_img_label_pair = {}
    test_img_label_pair ={}
    with open(label_file) as all_label:
        for label in all_label:
            label = label.rstrip()
            if label.startswith("train"):
                train_img_label_pair[label.split(" ")[0][:-4]+"_aligned.jpg"] = int(label.split(" ")[1])
            if label.startswith("test"):
                test_img_label_pair[label.split(" ")[0][:-4]+"_aligned.jpg"] = int(label.split(" ")[1])
    return train_img_label_pair, test_img_label_pair
```

Figure 7. Create label dictionary for data

Then transform images into numpy array and normalize them. Using Image class to open image and convert it to RGB format. Reshape it to 100*100*3 since it is colored 100*100 image. We scale these values to a range of 0 to 1 before feeding to the neural network model. Figure 5 shows the specific code for this process.

```
def load_to_numpy(img_label_pair, folder):

    length = 100
    width = 100
    limit = len(img_label_pair)
    labels = np.zeros((limit, 7))
    imgs = np.empty((limit, length, width, 3))

    i = 0
    for image_name in img_label_pair:
        img = Image.open(folder + image_name).convert('RGB')
        img = np.array(img).reshape((100,100,3))
        # important: normalize to [0,1]
        img = img/255
        # imgs = np.append(imgs, img, axis=0)
        imgs[i] = img # faster approach! learning algorithm is useful
        # labels[i] = img_label_pair[image_name]
        labels[i, img_label_pair[image_name]-1] = 1
        i += 1
    return imgs, labels
```

Figure 8. Data transformation and normalization

Finally, as shown in figure 6, call the label_loader function to create label pair dictionary. Furthermore, call load_to_numpy function (implemented in V2_helper.py) to create cleaned training and testing data, and save them to pickle file to accelerate the loading process next time.

```
# create image label pair as a dictionary
label_file = "../RAFDB/list_patition_label.txt"
folder = "../RAFDB/aligned/"
train_img_label_pair, test_img_label_pair = V2_helper.label_loader(label_file)
try:
    train_img, train_label, test_img, test_label = pickle.load(open("processed_data.pickle", 'rb'))
except:
    train_img, train_label = V2_helper.load_to_numpy(train_img_label_pair, folder)
    test_img, test_label = V2_helper.load_to_numpy(test_img_label_pair, folder)
    print("First time reading data, saving to pickle to speed up next time")
    pickle.dump((train_img, train_label, test_img, test_label), open("processed_data.pickle", 'wb'))
```

Figure 9. Save as pickle

4.2 MLP

For MLP, we defined our models in MLP_model.py file then import them in MLP_main.py file:

In this section, we will discuss and show how parameters, including optimizer, number of epochs, learning rate, dropout rate, batch size, and number of layers influence the

performance of MLP.

We create MLP models by assembling layers. In Keras, Dense is used to generate layers and provide output for next layer. As shown in figure 7, there are two layers in total. The number of neurons in hidden layer can be set as needed. However, the number of neurons in output layer must equal the number of class. In our project, it should be 7 because there are 7 kinds of emotions. The first layer of MLP requires the input dimension (shape), not including batch size. Therefore, we put 30000 because data are 100*100 colored images. Also we selected mini batch of 32 in this study for the reason that both the dimension (30000) of input and number of training image (12271) are not small. For transfer function, ReLU is used in hidden layer due to its sparsity and makes neural network run faster. Softmax is chosen for output layers because we aim to solve a classification problem.

```
def m_1():
    model = tf.keras.Sequential([
        layers.Dense(hidden1_num, activation='relu', input_dim=30000),
        layers.Dense(output_num, activation='softmax'),
    ])
```

Figure 10. Dense

Before training the model, a few more settings can be implemented by compile. Set loss to categorical crossentropy. Then Adam optimizer is selected in first model. Different optimizer will be compared below.

```
    model.compile(optimizer=tf.train.AdamOptimizer(0.001),
                  loss='categorical_crossentropy',
                  metrics=[tf.keras.metrics.categorical_accuracy])

    return model
```

Figure 11. Compile

Based on above statement, in the same condition (batch_size = 32, num_epoch = 5, num_layer = 2, hidden_num = 50, lr = 0.001), the accuracy of different optimizers is listed in table 1. SGD works better in this network. So, we select SGD.

Optimizer	Adam	Adadelta	RMSProp	SGD
Accuracy	22.1%	38.3%	38.5%	55.2%

Table 1. Optimizer

Next, just keep all the parameters but increase the number of epochs to 50, and add dropout with rate of 0.2 to prevent overfitting. The results are shown in table 2. It suggests that with the increase of epochs, the accuracy increases obviously.

lr	Dropout	10	20	30	40	50
0.001	0.2	0.5968	0.6196	0.6231	0.6349	0.6643

Table 2. Accuracy for different epochs_lr_0.001

In order to make the network converge faster, we consider changing learning rate larger to 0.01. The results are shown in table 3. Despite with expectation, larger learning rate cannot promote the performance of our model. It proves that step size of 0.01 is too big for this network that leads to skip the minimum points. So we keep the learning rate as 0.001.

lr	Dropout	10	20	30	40	50
0.01	0.2	0.5209	0.528			0.6379

Table 3. Accuracy for different epochs_lr_0.01

Adjusting mini batch size will influence model in many aspects. If we increase the size of mini batch, the iteration of one epoch will reduce, thus speed up the model for each epoch. To get the same accuracy with smaller batch size, larger size of mini batch needs more epochs. What's more, larger mini batch size requires larger memory capacity. We tried different batch size. According to results, the size of 32 is the best choice.

Batch size	32	64	128
Accuracy	0.6780	0.6670	0.6385
Time	123	132	137

Table 4. Accuracy for different batch size, lr_0.01

All the above discussion is based on a 2-layer perceptron. To get better performance, adding layer is another method. Keep the number of neuron in first layer as 50, adding layers to create different models as table 5. With the increase of layers, the accuracy declined. Therefore, 2-layer architecture is ideal for this data.

Hidden layer					Output layer	Accuracy
L_1: 50					7	0.5968
L_1: 50	L_2: 10				7	0.3862
L_1: 50	L_2: 10	L_3: 10			7	0.3862
L_1: 10	L_2: 10	L_3: 10	L_4: 10	L_5: 10	7	0.3862

Table 5. Accuracy for different number of layer, epoch=10

To summary, we adjusted certain parameters by controlling other parameters and got different types of MLP. In the result section, we will summarize and point out the best MLP for this dataset.

4.3 CNN

For convolution neural network, we defined our models in V2_models.py file then import them in V2_main.py file:

```
model = V2_models.first_model()
```

We used keras and tensorboard for real time model evaluation.

For the first model, we added two convolution layers with kernel size of 3 and the activation function ReLu. The filter sizes are 64 and 32. A fully connected layer is followed with the softmax classifier. We set the batch size of 32 and epoch number of 200. The validation accuracy of the first model is 65%. So a maxpooling layer is added to the second model to reduce spacial dimensions. However, with the same batch size and

epoch number, the validation accuracy only increased from 65% to 66%. Then we modified the filter size of two convolution layers to both 128. The deeper we go in the network, the smaller the spatial dimensions of our volume, and the more filters we learn. With larger filter size, the training time increased for each epoch. So we set the epoch number to 100. But the validation accuracy converged around 66% within 100 epochs. Then we considered to add dropout layers and one more fully connected layers to ensure we do not reduce our output size too quickly. The new fully connected layer is specified with a revised linear unit activation.

```
layers.Dropout(0.1),
layers.Flatten(),
layers.Dense(32, activation='relu'),
layers.Dropout(0.1),
```

The validation accuracy increased to 66.62% which is not that good as we expected. So we considered there might be something wrong with optimizer. We tried with four common optimizers Adam, Adadelta, RMSprop and SGD with learning rate decay and concluded with the top validation accuracy of 74%.

4.4 Light VGG

We tried to implement the original VGG16 at first, however the result is not very good. The networks take over 3 minutes to train one epoch and the size of the saved network is over 600MB.

Then we tried to reduce the number of convolution blocks, the number of kernels in each convolution block and the size of fully connected layer after the convolution blocks.

We came up with two light VGG style networks in the end.

Structure Name and Summary	
Light VGG V1	Light VGG V2
12s/epoch, validation_accuracy~74%	6s/epoch, validation accuracy~74%
Structure	
Block 1 In:100x100x3, Out:50x50x16	Block 1 In:100x100x3, Out:50x50x8
Conv2D (16 x (3x3))	Conv2D (8 x (3x3))
Conv2D (16x(3x3))	Conv2D (8x(3x3))
MaxPooling2D(2x2)	MaxPooling2D(2x2)

Block 2 In:50x50x16, Out25x25x32	Block 2 In:50x50x8, Out25x25x16(10000)
Conv2D(32x(3x3))	Conv2D(16x(3x3))
Conv2D(32x(3x3))	Conv2D(16x(3x3))
MaxPooling2D(2x2)	MaxPooling2D(2x2)
Block 3 In:25x25x32, Out:12x12x64(9216)	No Block 3
Conv2D(64x(3x3))	/
Conv2D(64x(3x3))	/
MaxPooling2D(2x2)	/
MLP: Shrink rate: ~4:1	MLP: Shrink rete:~16:1
Dense (2048)	Dense (512)
Dropout (70%)	Dropout (70%)
Dense (2048)	Dense (512)
Dropout (70%)	Dropout (70%)
Dense (512)	/
Dense (128)	/
Dense (32)	/
Dense (7) with SoftMax	Dense (7) with SoftMax

Table 6. Structure of Light VGG

5. Result

5.1 MLP

According to discussion in section 4, it is suggested that model "m_3" (can be found in python file called MLP_model.py) with batch size as 32 performances best for this dataset. As shown in following table 6, the accuracy reached 67.8%. The same model with batch size of 32 got a slightly smaller accuracy of 66.7%. Figure 9 and figure 10 shows the change of validation loss and validation accuracy. The blue line represents batch size 32 and red line stands for batch size 64.

Model	Batch_size	Num_epoch	layer	Num_hidden	Num_output	Dropout	lr	Optimizer	Accuracy
m_2	64	60	2	50	7	0.2	0.001	SGD	66.7%
	32	60	2	50	7	0.2	0.001	SGD	67.8%

Table 7. Ideal models

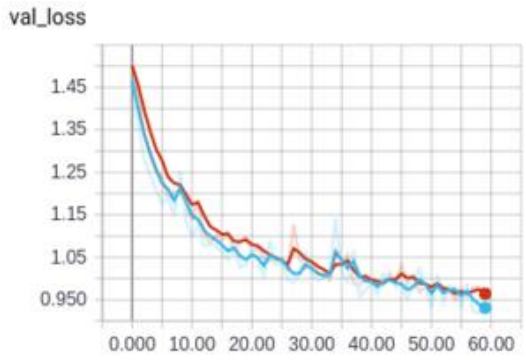


Figure 12 Validation Loss of MLP

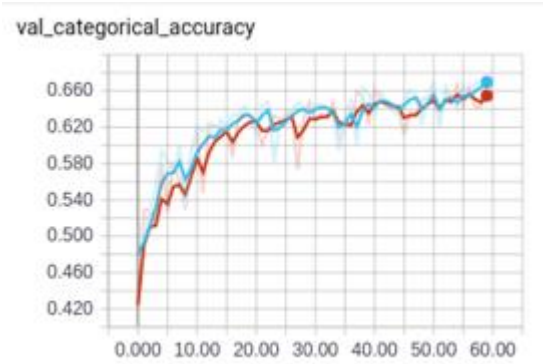


Figure 13 Validation Accuracy of MLP

5.2 CNN

During the first four models, we made some adjustments of model structures including adding maxpooling layer(2nd), modified filter size(3rd) and adding fully connected and dropout layers(4th).

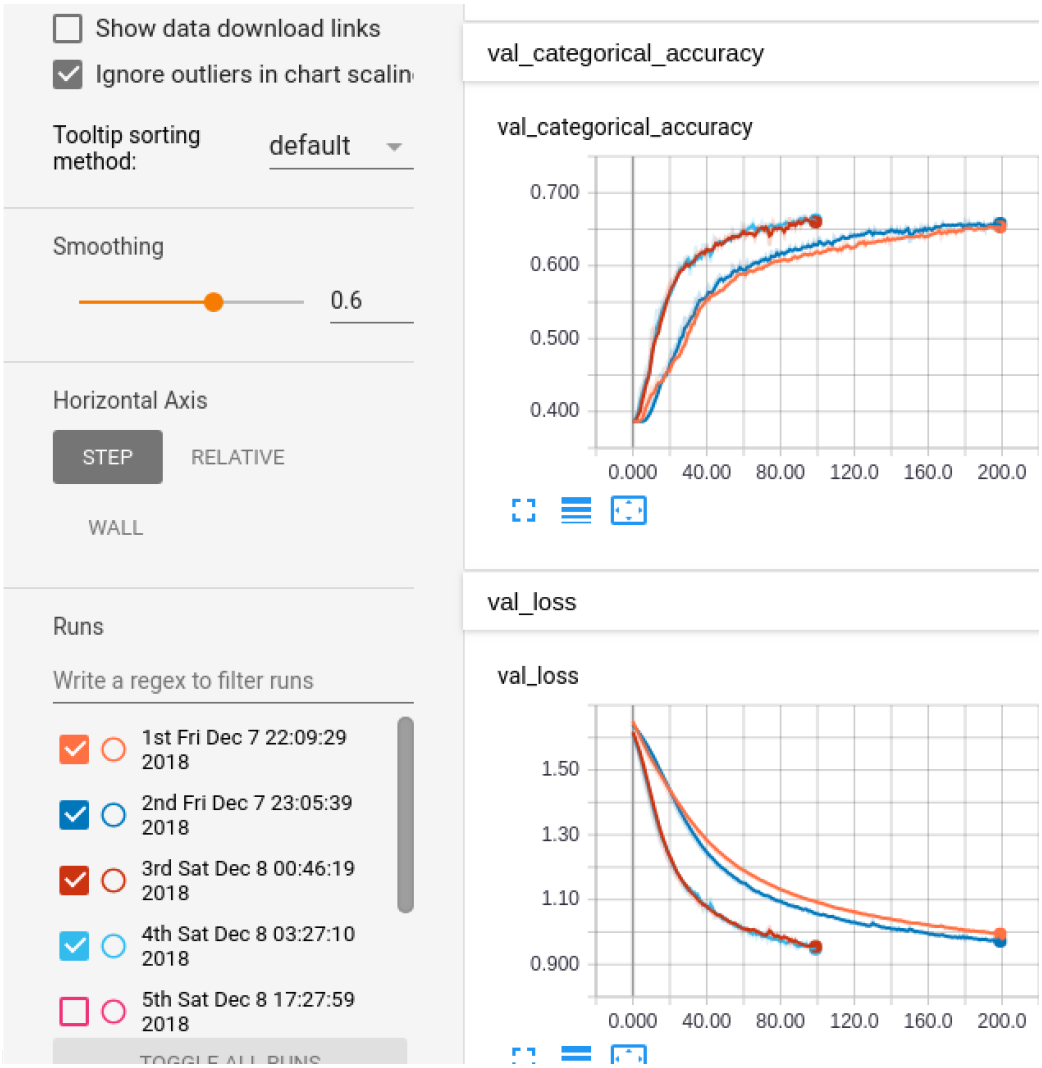


Figure 14 Validation Loss and Validation Accuracy of CNN

From the figure above, we could see the 4th model performance better than the other three with the validation accuracy around 66%. Then we take a look at the other four models which make adjustments of keras optimizer.

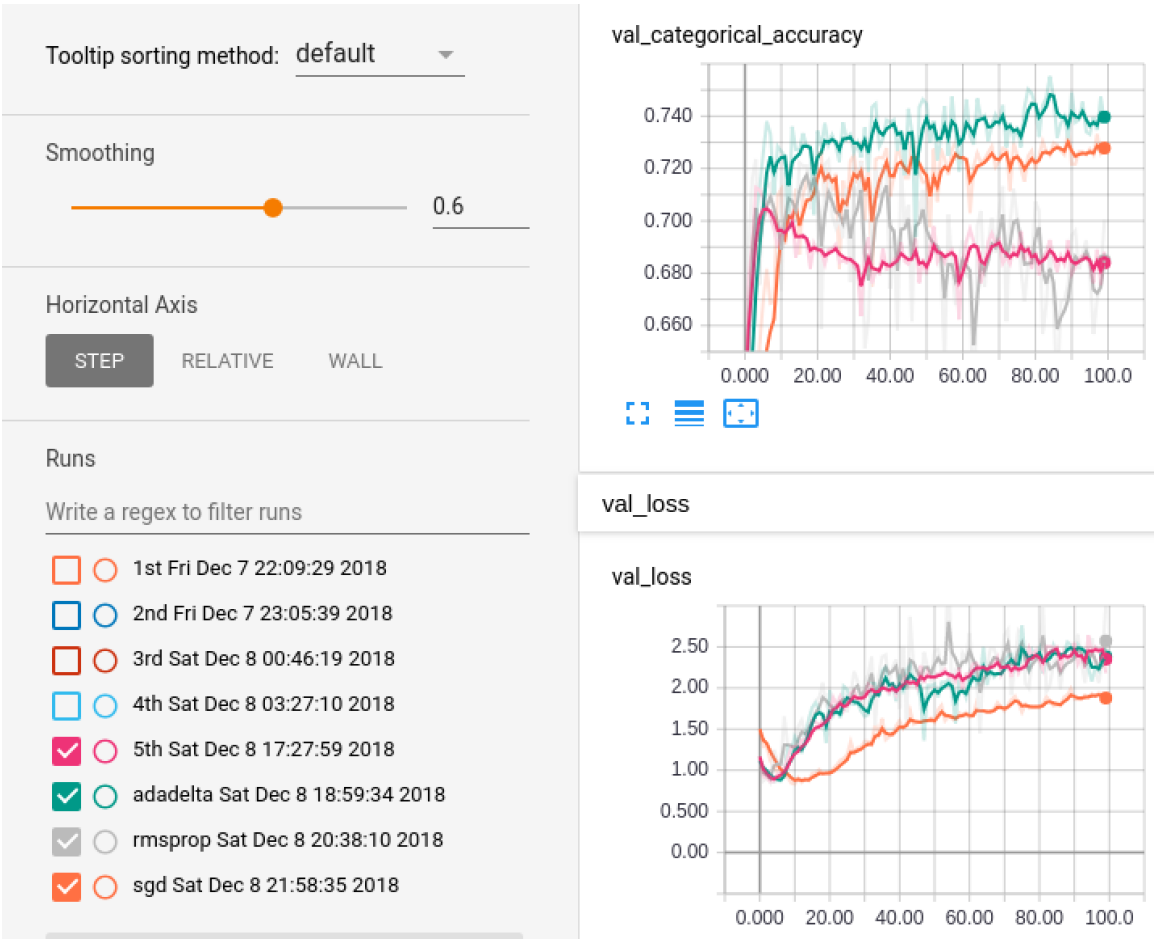


Figure 15 Validation Loss and Validation Accuracy of different Optimizer

With a learning rate decay of 1.0, the Adadelta optimizer performanced better than others in validation accuracy. While considering both validation loss and accuracy, we could see the performance of SGD optimizer is impressive as well from the figure above.

5.3 Light VGG

Both of the network can be trained in a short time(~3 minutes), but they still have the problem of overfitting. And the accuracy stuck at 74%, after which the loss of the validation set rises again and indicates overfitting. Thus, we need to apply early stop technique or check point the result of each epoch.

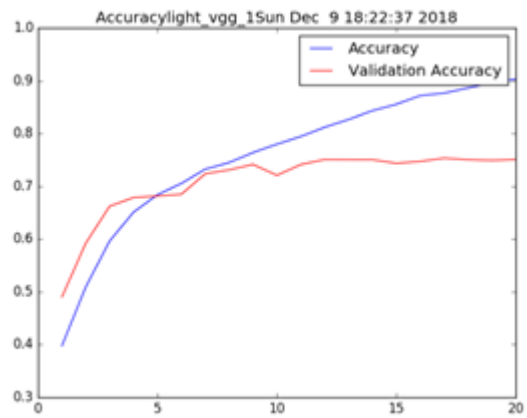
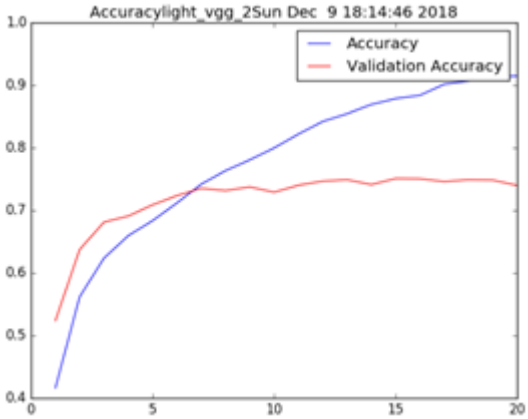
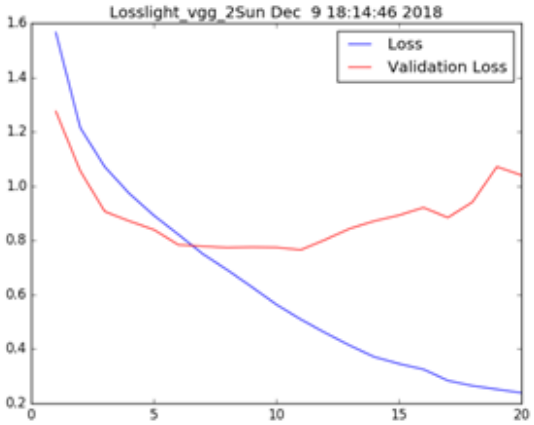
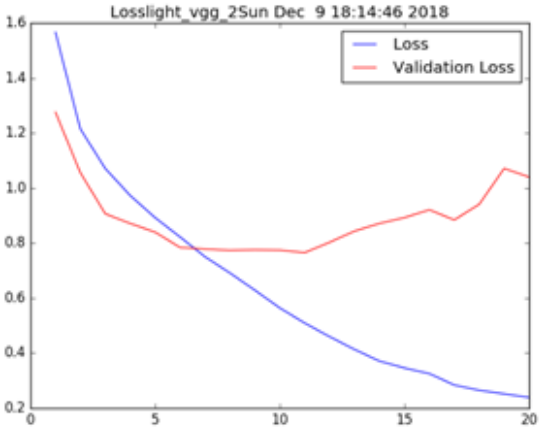
Light VGG V1	Light VGG V2
Accuracy	
	
Loss	
	

Table 8. Validation Accuracy and Validation Loss of Light VGG

6. Application: Real time facial expression recognition

We used the pre-trained HAAR Cascade with OpenCV3 to implement the real time facial expression recognition. (main program implemented in RTD_V2.py)
After the HAAR Cascade identify the location of the face, the program crops out the area of the face and reshapes it to (100x100x3) and feeds it to Keras model.

The model (implemented in RTD_helper.py) will return the predicted expression and it's score to the main program and print detail in the console. And the main program will draw a rectangle with the prediction label and score on the right corner.

However, the model cause perceptible delay in the program during the experiment on a laptop with 2.6GHz CPU. And it suggests that the model still needs improvement on speed to make a industry level product. (e.g. an app on the smartphone)

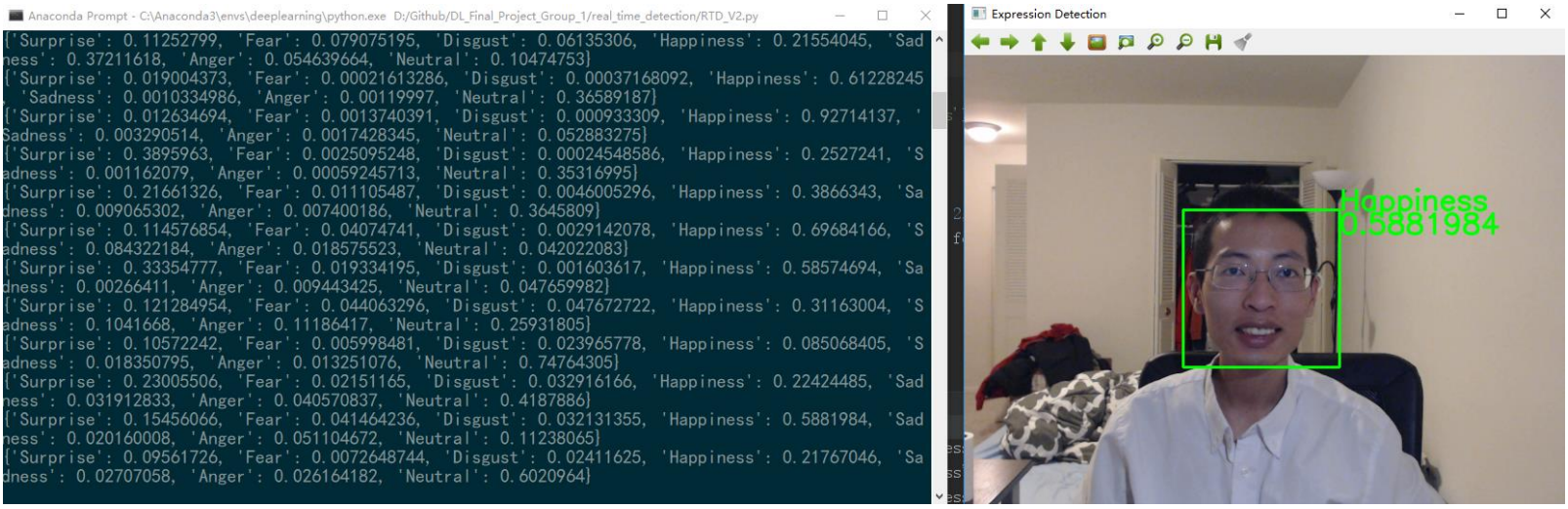


Figure 16 GUI of real time facial expression recognition

7. Summary and conclusion

We have tried Multi-Layer Perceptron model and Convolution Neural Network in this project. The result suggests that MLP is not a good model for expression classification on RAF-DB, as the accuracy is only 67.8. Which has been consistent with the conclusion of many computer vision researchers.

One of the method to classify RAF-DB is to build a CNN from scratch and doing experiment on the structure and optimizers. And another is to learn from the common CNN structures and adopt their design method.

However, in this project, both method did not generate a perfect model. The accuracy is around 74%, and both models has unimprovable overfitting issue.

Despite the same accuracy and overfitting issue, light VGG V2 is our best model for RAF-DB single label subset because it's faster and smaller thus more suitable for real time facial expression detection.

After doing some literature review, the future step could be collecting more data, pretrain the model on other dataset (e.g. ImageNet) and use transfer learning.

We also made a prototype program of real time facial expression recognition to make the outcome of this project more practical instead of just comparing the accuracy and loss. We learnt that while the forward propagation in training can be very fast with CUDA on

a powerful cloud VM, we still need to improve the model and reduce the time complexity to make it an industry level product.

8. References

1. Li, S., & Deng, W. (2018). Deep Facial Expression Recognition: A Survey. arXiv preprint arXiv:1804.08348.

2. Ekman, P., & Friesen, W. V. (1971). Constants across cultures in the face and emotion. Journal of personality and social psychology, 17(2), 124.

Li, S., & Deng, W. (2019). Reliable crowdsourcing and deep locality-preserving learning for unconstrained facial expression recognition. IEEE Transactions on Image Processing, 28(1), 356-370.

3. Li, S., Deng, W., & Du, J. (2017, July). Reliable crowdsourcing and deep locality-preserving learning for expression recognition in the wild. In Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on (pp. 2584-2593). IEEE.

4.Li, S., & Deng, W. (2019). Reliable crowdsourcing and deep locality-preserving learning for unconstrained facial expression recognition. IEEE Transactions on Image Processing, 28(1), 356-370.

5. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

6. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).